Intro to Haskell

Following along

```
$ vim Haskell.hs

# in another session

$ ghci Haskell.hs

# edit away in Haskell.hs

*Main> :r
Ok, one module loaded.
*Main>
```

- No variables
- No for loops
- No if statements
- No pointers or classes
- Your program is one big expression

```haskell
myName :: String
myName = "Luke"
```

```haskell
foo :: Int -> Int
foo x = x + 42
```

```haskell
bar x y :: Int -> Int -> Int
bar x y = (foo x) - y
```

```haskell
length :: String -> Int
length str = 1 + length remaining
    where remaining = tail str
    -- tail "abcd" = "bcd"
```

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

```haskell
bar :: Int -> (Int -> Int)
bar x y = (foo x) - y

bar' :: Int -> Int
bar' = bar 3

bar'' :: Int
bar'' = bar' 7
```

Currying 

Polymorphism (Generics)

```haskell
reverse :: [a] -> [a]
reverse (x:xs) = reverse xs ++ [x]
```

```haskell
-- A constraint that a must be “orderable”
sorted :: Ord a => [a] -> Bool
sorted [] = True
sorted [x] = True
sorted (x:y:xs)
  | x <= y = sorted (y:xs)
  | otherwise = False
```

How do you do control flow?

```haskell
describeNumber :: Int -> String
describeNumber x = if x > 100
then "That's big!"
else "It's pathetically small"
```

What about a for loop?

```haskell
for :: (a -> b) -> [a] -> [b]
for f [] = []
for f (x:xs) = [f x] ++ for f xs
```

Curried

```
> for (* 2) [1, 2, 3]
[2, 4, 6]
```

```
> for not [True, True, False]
[False, False, True]
```

```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
map :: (a -> b) -> [a] -> [b]

filter :: (a -> Bool) -> [a] -> [a]
```

```
> filter even [1, 2, 3, 4, 5]
[2, 4]
```

Get all primes

```
isPrime n = not (any (divisible n) [2..(n - 1)])
divisible x y = x `mod` y == 0
```

```
> filter isPrime [3,6,7,9,11]
[3,7,11]
```

```
> filter isPrime [2..]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61
67,71,73,79,83,89,97,101,103,107,109,113,127,131,1
7,139,149,151,157,163,167,173,179,181,191,193,197,
99,211,223,227,229,233,239,241,251,257,263,269,271
277,281,283,293,307,311,313,317,331,337,347,349,35
```

Laziness

```
> head (filter (> 9000) (filter isPrime [2..]))
9001
```

Why do computers fix themselves whenever you turn them off and on again?

State is bad

```
int add(int x, int y);
```

```
int i = 0;
int add(int x, int y) {
    i++;
    if (i == 1000) return 4329781238912389;
    return x + y;
}
```

Purity

```
add :: Int -> Int

add x y = x + y
  if ??? == 1000
    then 123456
    else x + y
```

But then how can it be useful?

```
getChar :: Char

getChar == 'a'
getChar == 'b'

-- function must return the same
thing for the same arguments!
```

But then how can it be useful?

I have side effects!

```
getChar :: IO Char
```

```
getSomeChars :: IO ()
getSomeChars = do
    foo <- getChar
    bar <- getChar
    baz <- getChar
    putStrLn [foo, bar, baz]
```

Run getChar and store it in foo

foo :: String

Word reverser

```haskell
-- dirty code
main :: IO ()
main = do
    str <- getLine
    let result = reverseWords str
    putStrLn result

-- pure code!
reverseWords :: String -> String
reverseWords str = unwords (reverse (words str))
```

Let’s write cat

```haskell
import System.Environment
main :: IO ()
main = do
     [file1, file2] <- getArgs
     a <- readFile file1
     b <- readFile file2
     putStrLn (a ++ b)
```

```
$ ghc Cat.hs
[1 of 1] Compiling Main ( Cat.hs, Cat.o )
Linking Cat ...
$ ./Cat foo.txt bar.txt
```

Data types

```haskell
data MyType = Foo Int
            | Bar String Int
            | Baz Bool Char

x :: MyType
x = Foo 32

y :: MyType
y = Bar "asdf" 1234
```

Record syntax

```haskell
data Person = Person
  { name          :: String
  , age           :: Int
  , likesHaskell :: Bool
  }
```

```haskell
luke :: Person
luke = Person
  { name = "Luke"
  , age  = 21
  , likesHaskell = True
  }

> likesHaskell luke
True
```

Built in data types

```haskell
data Maybe a = Just a | Nothing
```

```haskell
first :: [a] -> Maybe a
first [] = Nothing
first (x:xs) = Just x
```

Built in data types

```haskell
data Either a b = Left a | Right b
```

```haskell
first :: [a] -> Either String a
first [] = Left "The list is empty"
first (x:xs) = Right x
```

Handling Data Types

```haskell
case first ["Hello", "world"] of
    Left msg -> putStrLn msg
    Right x -> putStrLn x

myFunc :: Maybe Int -> Int
myFunc Nothing = 0
myFunc (Just x) = x
```

Built in data types

```
data List a = Nil | Cons a (List a)
```

Data types

```haskell
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)

smallTree :: Tree Int
smallTree = Leaf 42

bigTree :: Tree Char
bigTree = Branch (Branch (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

```haskell
data List a = Nil | Cons a (List a)

map :: (a -> b) -> [a] -> [b]
```

```haskell
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)

smallTree :: Tree Int
smallTree = Leaf 42

bigTree :: Tree Char
bigTree = Branch (Branch (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

```haskell
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x) = Leaf (f x)
treeMap f (Branch x y) = Branch (treeMap f x) (treeMap f y)
```

```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
treeMap :: (a -> b) -> Tree a -> Tree b
```

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

```haskell
instance Functor List where
    fmap f [] = []
    fmap f (x:xs) = [f x] ++ fmap f xs

instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch x y) = Branch (fmap f x) (fmap f y)

instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

Things that can be mapped

Functor

Things that can be sequenced

Monad

Haskell is really good at abstraction!

Things that can be appended

Monoid

Things that can be equal

Eq

Things that can be ordered

Ord

Monad

Things that can be sequenced

IO is a Monad

```haskell
getChar :: IO Char
readFile :: String -> IO String
putStrLn :: String -> IO ()

-- the do block signifies that these actions
-- are sequenced together, since the order
-- in which these are carried out matters!

main :: IO ()
main = do
    myChar <- getChar
    a <- readFile "Hello"
    putStrLn a
```

Maybe is a Monad

```
hopefullyThisWorks :: Maybe Int
riskyCalculation :: Int -> Maybe Int
fingersCrossed :: Int -> Int -> Maybe Int
```

```
mightFail :: Maybe Int
mightFail = do
    x <- hopefullyThisWorks
    y <- riskyCalculation x
    fingersCrossed x y
```

[] is a Monad

```haskell
-- possible sums that could be made
-- when rolling a die
rollDie :: Int -> [Int]
rollDie sum = map (+ sum) [1..6]

possibleSums :: [Int]
possibleSums = do
    firstSum <- rollDie 0
    secondSum <- rollDie firstSum
    rollDice secondSum


> possibleSums
[3,4,5,6,7,8,4,5,6,7,8,9,5,6,7,8,9,10,6,7,8,9,10,11,7
10,11,12,8,9,10,11,12,13,4,5,6,7,8,9,5,6,7,8,9,10,6,7
10,11,7,8,9,10,11,12,8,9,10,11,12,13,9,10,11,12,13,14
7,8,9,10,6,7,8,9,10,11,7,8,9,10,11,12,8,9,10,11,12,13
```

Learn Haskell

- Learn You A Haskell for Great Good
  learnyouahaskell.com
- School of Haskell
  schoolofhaskell.com
- Try out the millions of packages on
  hackage.com
  - turtle package is a good tutorial